Safety-critical Java for embedded systems

Schoeberl, Martin; Dalsgaard, Andreas Engelbredt; Hansen, René Rydhof; Korsholm, Stephan E.; Ravn, Anders P.; Rios Rivas, Juan Ricardo; Strøm, Torur Biskopstø; Søndergaard, Hans; Wellings, Andy; Zhao, Shuai

Published in:
Concurrency and Computation: Practice & Experience

Link to article, DOI:
10.1002/cpe.3963

Publication date:
2016

Document Version
Peer reviewed version

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Safety-Critical Java for Embedded Systems

Martin Schoeberl,* Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Stephan E. Korsholm, Anders P. Ravn, Juan Ricardo Rios Rivas, Tórur Biskopstø Strøm, Hans Søndergaard, Andy Wellings, and Shuai Zhao

Technical University of Denmark, Aalborg University, Denmark, University of York, UK

SUMMARY

This paper presents the motivation for and outcomes of an engineering research project on certifiable Java for embedded systems. The project supports the upcoming standard for safety-critical Java, which defines a subset of Java and libraries aiming for development of high criticality systems. The outcome of this project include prototype safety-critical Java implementations, a time-predictable Java processor, analysis tools for memory safety, and example applications to explore the usability of safety-critical Java for this application area. The text summarizes developments and key contributions and concludes with the lessons learned.

Received . . .

KEY WORDS: safety-critical Java, real-time systems, embedded systems

1. INTRODUCTION

Modern household appliances are examples of products with embedded software. Some of this software implements safety functions, for instance ensuring safe shutdown when there is a risk of overheating; therefore it must undergo certification. Certification of software is thus a topic that attracts considerable attention as witnessed by more recent safety standards. It is clear that the standards do not prescribe any particular programming language or any specific tools, but nevertheless they outline a development process and associated documentation that stresses the use of safe features of the language used, and the use of good tools to implement the final code as well as performing analyses and tests.

Java is a *safer* language than e.g., C with stronger type checks at compile time and avoidance of raw pointers. Furthermore, the first Java specification request was for the Real-Time Specification for Java (RTSJ) [1]. Therefore, Java has been considered as a language for real-time and safety-critical systems. Especially a restricted version of Java [2], called Safety-Critical Java (SCJ), is an approach of using Java in future safety-critical systems.

With this in mind we embarked on an effort to understand what would be needed to build trustworthy systems. In summary, we found that there shall be a dependable platform (including the hardware and the software runtime), and a development environment that binds the profile and the platform together and provides analyses that provides evidence that can be used as part of a safety case. These have been the objectives of the Certifiable Java for Embedded Systems (CJ4ES) project.

From a research angle, the CJ4ES project was seen as a way to consolidate and integrate a number of results from previous work by members of the team. Key previous results include development of

*Correspondence to: Department of Applied Mathematics and Computer Science, Technical University of Denmark, Richard Petersens Plads, Building 322, room 128, 2800 Lyngby, Denmark

a time-predictable Java processor [3, 4], contributions to profiles for predictable Java [5, 6, 2], and development of analysis tools [7, 8] as well as concepts for integration of the tools. As most of the previous results are open-source,[†] the results from this project are available as open-source as well.

Parts of the operational work plan of the project were to evaluate SCJ by implementing it and using it in example applications. Within the project we have explored the standard, provided two implementations of it, built example applications, investigated the programmability of SCJ, and last but not least, given feedback to the expert group of JSR 302.

This paper is organized in 10 sections. The following section presents related work and Section 3 provides background on safety-critical Java. Section 4 presents two Java virtual machines (JVMs), the Java processor JOP and the hardware near VM (HVM), used as platforms for our SCJ implementations, while Section 5 describes the two SCJ implementations. Section 6 presents hardware support for SCJ within JOP. Section 7 describes two tools: the memory safety analysis tool and the worst-case memory consumption analysis tool. Section 8 explores the expressiveness of SCJ with implementations of applications. Section 9 evaluates the results of the project and describes lessons learned. Section 10 concludes the paper.

A significant effort has been on a test suite for the SCJ. This work is reported in a separate extended paper.[‡]

This paper is an extension of a paper published at JTRES 2014 [9]. The extensions include a new section on lessons learned, an implementation of SCJ level 2, and more details on the memory safety analysis and the RepRap use case. Other sections have been updated.

2. RELATED WORK

Kelvin Nilsen drove early work on Java for real-time systems with the proposal of the PERC Real-Time API [10] and the NIST paper on real-time extensions for Java [11]. A consortium under lead by Greg Bollella defined the Real-Time Specification for Java (RTSJ) [1]. With the RTSJ also the Java Community Process was initiated, thus making RTSJ the first Java Specification Request (JSR 1). RTSJ has been used for implementing industrial scale systems. However, as a pioneering work, it had some features that were very dynamical and thus hard to justify in critical systems where memory consumption and timing have to be predictable. Consequently, work on a simplification and rationalization of RTSJ started within the Java community process with a specification of *Safety-Critical Java* (SCJ). It is now proceeding in close cooperation with a revision of RTSJ: RTSJ 2.0.

Puschner and Wellings presented the first proposal for a simplified RTSJ [12]. In particular, they introduced the concept of an *initialization* and a *mission* phase into Java for real-time systems. In the initialization phase, all threads, event handlers, and shared objects are set up. During the mission phase the threads and event handlers are scheduled. The restrictions, e.g., static priorities, no call of sleep, no `wait/notify`, and no dynamic class loading, have been taken up by the SCJ expert group for the SCJ specification. Later, the proposal of Puschner and Wellings was refined and renamed to Ravenscar Java [13], emphasizing the heritage of the concepts from the Ada Ravenscar tasking profile [14].

Due to the additional complexity introduced by inheriting directly from RTSJ classes, later proposals for a safety-critical Java profile argue for an API that is independent of the RTSJ [15, 5]. This would have the further advantage of avoiding possible programmer confusion at having two versions of the RTSJ classes (the original and the restricted). This problem stems from the fact that RTSJ is more expressive than SCJ, but SCJ classes extend the RTSJ classes. This results in a situation where the RTSJ classes in the SCJ version have to be restricted. To model this *inverse* relation between RTSJ and SCJ it has been proposed to build the class hierarchy the other way round: RTSJ shall inherit from classes as defined in SCJ [6].

Søndergaard et al. provide an implementation of the Ravenscar Java profile [16]. The implementation targets industrial applications and uses an aJ-100 processor developed by aJile

[†] see `https://github.com/jop-devel/jop` and `http://www.icelab.dk/download.html`
[‡] Submitted to the same special issue.

DOI: 10.1002/cpe

Systems [17]. The aJ-100 is a 32-bit microprocessor that directly executes Java bytecodes (implemented in microcode) as its native instruction set. In addition, it provides a microcode programmed real-time kernel that provides, among other things, support for scheduling, context switching and object synchronization.

Plsek et al. present one of the first implementations of SCJ on an embedded platform [18]. They provide an implementation of SCJ's level 0 running on the OVM virtual machine [19]. The OVM is a framework that enables alternate implementations of core VM functionality, e.g., different versions of priority inheritance monitors in order to build and test VMs with different features. OVM uses an ahead-of-time compiler to translate Java code to C++ and then it uses the GCC compiler to obtain machine code. SCJ's implementation on OVM runs on an FPGA board executing the RTEMS real-time operating system on a LEON3 processor.

In the work presented by Bøgholm et al. a different approach is taken [6]. Instead of using the classes defined by the SCJ profile, their implementation is based on a profile called Predictable Java (PJ). PJ is a Java profile suitable for the development of high-integrity real-time embedded systems. It is inspired by and builds upon previous work on Java profiles, most notably the Ravenscar profile for Java [16, 12, 13, 5]. The profile is based on the execution of event handlers grouped in missions, which in turn are also considered event handlers.

Originally it was planned that PERC Pico [20] would be SCJ compliant. However, Atego later considered implementing SCJ in addition to the current PERC Pico notion of safety-critical Java. The intention was to support both APIs and memory models in a single JVM [21]. The later paper also describes the differences between the SCJ memory model and the PERC Pico memory model. However, after TPC acquired Atego, development on PERC Pico ceased.

3. SAFETY-CRITICAL JAVA

Java specification request number JSR 302 for Safety-critical Java Technology (SCJ) [22] targets platforms for safety-critical real-time systems, with special focus on systems that require formal certification. One constraint of SCJ was that the reference implementation could execute on top of RTSJ. This resulted in a definition where SCJ extends RTSJ classes and interfaces and then disallow certain more dynamic features. The work on SCJ has been significantly delayed by this decision, because the RTSJ specification is under revision in an overlapping JSR 282 activity. Apart from the benefit of reusing the RTSJ experience, the Reference Implementation (RI) can build on existing RTSJ work. This is, however, not recommended for operational platforms that need to be certified, since much disallowed RTSJ code has to be removed.

An SCJ application consists of *missions* that contain a fixed number of cooperating sequential processes, the *managed schedulable objects* (MSO). Each mission goes through three phases: initialization, mission, and cleanup. A mission represents a mode of operation for an embedded program, and the MSO's are the real-time activities. A MSO is a either a periodic, an aperiodic handler, or, in level 2, a real-time thread. Periodic handlers are run periodically, where aperiodic handlers are released by other handlers or by external events like interrupts.

A mission goes through a sequential initialization phase first to initialize and register the MSOs. Then it enters an active phase, where the MSOs run, and, if it ends, finishes with a cleanup phase, where the MSOs are removed and their resources are released.

In order to ease certification of SCJ programs, for different application areas, three *levels* of conformance are defined to support different levels of application complexity

Level 0 Here a sequence of missions is run. A mission consists of periodic handlers only, and in the active phase a cyclic executive schedules them statically.

Level 1 Also here, a sequence of missions is run. However, a mission may include aperiodic handlers (although SCJ assumes they will be sporadic in natures, it does not enforce a minimum interarrival time between releases; therefore, it uses the term aperiodic), and in the active phase fixed priority preemptive scheduling is used. Thus interrupt driven handlers

DOI: 10.1002/cpe

are permitted. The preemptive scheduling means that a priority ceiling protocol has to be used for synchronized access to objects shared among handlers.

Level 2 This allows missions to be nested, so they can be run concurrently. Also, a mission may include real-time threads. Whereas a handler has application logic without self-suspension, a real-time thread uses explicit self-suspension in its application logic.

3.1. Memory Organization

Dynamic memory allocation, in any form, is considered with some caution by standards for safety-critical applications. However, to enable some form of dynamic memory allocation, SCJ uses a restricted version of the RTSJ scoped memory concept. Objects are allocated in a memory area with a defined lifetime.

All missions share a global *immortal memory*. Objects allocated here live for the entire lifetime of the application. When a mission is initialized, objects are allocated in a fresh *mission memory*. These objects are shared among the MSOs of the mission and live until the end of the mission's cleanup phase. MSOs have a *private memory* that lives as long as the application logic of the MSO is active. It is intended for objects that are used in the algorithms of this logic. These objects cannot be shared with other MSOs or be linked to structures in other memory areas. Should a MSO have complex computations, it can allocate nested private memories which contain objects for these, possibly nested, computations. The lifetime of these nested private memories is the lifetime of a computation encapsulated in a run method.

The size of the memory areas of the different kinds must be declared by the application. Since it is far from easy to give tight estimates for these, a tool is presented in Section 7.2.

In order to implement an SCJ application, an implementation of the `Safelet` interface must be proviced. An implementation of the `Safelet` interface must provide a method `getSequencer` that returns a mission sequencer object, that produces the sequence of missions. This interface will also declare the size of immortal memory.

3.2. The Memory Model

The memory areas that are visible to a MSO are structured as a simple *stack* of scopes. In this scope stack recently allocated and shorter-lived memory areas can be found near the top and longer-lived memory areas are located nearer the bottom of the stack. Only the mission and immortal memory areas are shared between MSOs. This is in contrast to the full scoped memory model of the RTSJ, where more general relationships between schedulable objects and memory areas can be constructed.

Programmer-managed memory areas introduce the risk of *dangling references*. A dangling reference is the result of (unintentionally) deallocating a memory area where there are accessible (live) objects. In order to avoid dangling references, the SCJ memory model dictates that reference assignments are allowed only when the reference points to an object in a memory area *deeper* in the stack. This is difficult to handle in an application, therefore Section 7.1 describes a tool to check that a program does not contain any (potentially) dangling references.

4. JAVA VIRTUAL MACHINES FOR SCJ

We have developed and used two Java virtual machines (JVM): (1) the Java optimized processor (JOP) [4] and (2) JVM implemented in software, called Hardware near VM (HVM) [23]. JOP is implemented in a field-programmable gate array (FPGA), whereas HVM targets embedded processors with a small memory budget. Both JVMs support *hardware objects* for I/O access and first level interrupt handlers [24]. The scheduler for the JVMs is simply a first level interrupt handler written in Java.

DOI: 10.1002/cpe

4.1. The Java Optimized Processor

JOP is an implementation of the JVM in hardware [4]. To obtain deterministic and well-known execution time for bytecode instructions, JOP uses microcode as its native instruction set, with most of the JVM bytecodes implemented as a sequence of microcode instructions. Only the more complex bytecodes, e.g., the `new` and `newarray` bytecodes, are implemented in Java. The mapping of bytecodes to microcode instructions is done using a translation stage that converts a bytecode instruction into the start address of the microcode sequence that implements that bytecode. This translation stage takes exactly one clock cycle and thus it can be pipelined.

JOP is a RISC style processor implementing a stack architecture. It is designed to be time-predictable and to be used in hard real-time systems. Since standard caches are notoriously difficult to analyse, e.g., in connection with worst-case execution time (WCET) analysis [25], the caches in JOP have been designed to be time-predictable and to simplify WCET analysis. In total, JOP includes three such caches: (1) a stack cache [26] that caches local variables and the operand stack, (2) a method cache [27] instead of an instruction cache that caches whole methods, and (3) an object cache [28] to cache objects. The following discussion explains in more detail how these caches are designed to be time-predictable and thereby facilitate WCET analysis.

The method cache is a replacement for an instruction cache found on other architectures. Lookup in the method cache can result in a miss only on method invocation or on a return from a method. All other bytecode instructions are guaranteed hits, vastly simplifying WCET analysis and thus requiring less state to be tracked for a WCET analysis. The WCET analysis for the method cache uses a scope based approach where program scopes are grown larger up to the point where all methods within a scope fit into the method cache. Calls to those methods can then only miss once within that scope [29].

As the JVM is a stack machine, the stack is used for operands and for local variables. Therefore, basically all bytecodes access data in this memory area up to three times per instruction. In the interest of performance, the stack memory must be cached. Since most stack accesses are concerned only with the top elements of the stack, the JOP stack cache is split into two components: the two top elements are implemented by two explicit registers to enable two reads and one write access to the top elements in a single clock cycle. These two registers are backed up by an on-chip memory. The on-chip memory must be sized big enough to hold *all* stack data for a thread. There is no automatic spilling or filling of the stack cache, further simplifying WCET analysis since stack accesses are cache hits by design.

More challenging is the caching and WCET analysis of objects, which are allocated on the heap. The addresses of these objects are only known at runtime making standard data cache analysis impossible. Therefore, we have developed a so-called object cache that caches complete objects [28]. The cache has a high associativity that enables WCET analysis to track objects in the cache symbolically [30].

The well-known execution time of individual bytecodes and the special caches simplify WCET analysis. JOP is the only JVM that includes a WCET analysis tool in the source distribution [8]. This combination of a time-predictable processor and the included WCET analysis tools makes JOP a good execution platform for safety-critical applications.

JOP is a multicore processor where the number of cores is configurable. A time-division multiplexing memory arbiter enables WCET analysis even of a multicore processor [31]. Using multicores in SCJ is defined for level 1 and level 2 of SCJ. To avoid issues with lock contention on a multicore, level 0 is defined only for a single core. However, the synchronized access to shared resources can be guaranteed when incorporated in the static schedule of a cyclic executive [32]. Therefore, a cyclic executive is an interesting alternative on a multicore to just using single core processors in SCJ level 0.

Although JOP incorporates a number of experimental and state-of-the-art features, the platform has been shown to be mature enough to be used in industrial applications [33]. Besides the WCET analysis tool WCA [8], the JOP distribution also contains an optimizer [34] for method inlining at bytecode level. As this optimizer works at bytecode level, it is not JOP specific and can also be used by other JVMs.

DOI: 10.1002/cpe

4.2. The Hardware Near Virtual Machine

The Hardware near Virtual Machine (HVM) [23] is a JVM intended for resource constrained platforms. The design of the HVM is inspired by existing C programming environments for embedded systems, e.g., GCC. Some important features of C, that the HVM seeks to support for Java as well, are:

- *Proportionality*: The memory footprint of a compiled C executable is proportional to the size of the source code written and libraries used. The compiler and linker strip away dead code and do not require the presence of libraries not used. This results in very tight executables in terms of code memory requirements
- *Efficiency*: C compilers produce very efficient and highly optimized code
- *Hardware near*: C programs can access device registers and memory directly. They can also program 1st level interrupt handlers and place variables in specific sections of memory.

The HVM build environment performs a whole-program static analysis and calculates a conservative estimate of all classes and methods that may be entered at runtime. This set of classes and methods is called the dependency extent. Starting from the main entry point of the program the dependency extent is calculated by computing a conservative estimate of all possible traces through the program. For control flow branches like `if`, the conditional and both the `else` and `then` branches are added to the extent. This method is straightforward and clearly a safe conservative estimate of the dependency extent. Opposed to this simple case, predicting the flow of control for a virtual method invocation is not straightforward. To solve this issue the static analysis keeps track of all possible classes that may have been instantiated along all traces leading up to the method invocation. Knowing this set enables the analysis to predict the possible targets of the method invocation. This may require revisiting the same call site until a fixed point is reached. This method is an instance of the well-known abstract interpretation framework [35].

To achieve efficiency, the HVM supports ahead-of-time (AOT) compilation of Java bytecode straight into C, as well as supporting standard interpretation. The programmer can select which methods to interpret and which to compile into C. Both execution styles are supported since they each have their strengths and weaknesses: interpretation tends to be slow, but yields a smaller code memory footprint, whereas AOT compiled code tends to be significantly faster but yields a larger code memory footprint. To enable hardware near programming the HVM supports (1) hardware objects, (2) native variables, and (3) first level interrupt handlers. Hardware objects [36] can be viewed as a way for the programmer to control the memory address at which an object gets allocated. This can be used to map object fields to device registers. Native variables are a way to map C symbols and C names to static Java variables. This provides direct access from Java space to symbols and macros defined in the C runtime system and linked C libraries. Native variables can in many cases be used as an alternative to hardware objects. First level interrupt handlers are usually defined in existing C environments by adding compiler attributes or following specific naming schemes when defining the function header. This is supported by the HVM through the use of Java annotations: static Java methods can be annotated as interrupt handlers and the annotation contains fields to describe how the AOT compiler should translate the method signature into C. An example of a UART transmit interrupt for the AVR architecture is included in Figure 1.

In this example the static variable `UART0_DATA` is annotated as an `IcecapCVar`. This makes the AOT compiler generate C code that accesses that variable directly as it is declared in C space, and not as a static class variable as would normally be the case. Also the method `uartr0_transmit_interrupt` is annotated as an `IcecapCFunc` with the field `signature` set to `ISR(UART0_TRANSMIT_INTERRUPT)`. This makes the AOT compiler translate it into a C function with that signature. For the AVR embedded architecture this identifies the function as the interrupt service routine for the UART transmit interrupt.

Furthermore, the HVM has been used as an execution environment to experiment with WCET and schedulability analysis [37, 38]. Furthermore, the development of a Technology Compatibility Kit (TCK) for SCJ level 0 and level 1 runs on top of the HVM.

DOI: 10.1002/cpe

```
static byte UART_TxHead;
static byte UART_TxTail;
static byte UART_TxBuf[];

@IcecapCVar(requiredIncludes = "#include \"avr/io.h\"")
static byte UART0_DATA;
@IcecapCVar(requiredIncludes = "#include \"avr/io.h\"")
static byte UART0_CONTROL;
@IcecapCVar(requiredIncludes = "#include \"avr/io.h\"")
static byte UART0_UDRIE;

@IcecapCFunc(signature = "ISR(UART0_TRANSMIT_INTERRUPT)",
             requiredIncludes = "#include <avr/interrupt.h>")
private static void uartr0_transmit_interrupt() {
   byte tmptail;

   if (UART_TxHead != UART_TxTail) {
      // calculate and store new buffer index
      tmptail = (byte) ((UART_TxTail + 1) & UART_TX_BUFFER_MASK);
      UART_TxTail = tmptail;
      // get one byte from buffer and write it to UART
      UART0_DATA = UART_TxBuf[tmptail]; // start transmission
   } else {
      // tx buffer empty, disable UDRE interrupt
      UART0_CONTROL &= ~(1 << (UART0_UDRIE));
   }
}
```

Figure 1. AVR transmit interrupt written in Java

5. SAFETY-CRITICAL JAVA IMPLEMENTATIONS

We have implemented the SCJ infrastructure on top of two JVMs: JOP and the HVM. Those two SCJ implementations have been done independently, just sharing concepts on implementation of the scoped memory hierarchy and usage of first level interrupt handlers.

We implement the scoped memory model as defined by SCJ and briefly described in Section 3. For our implementations of the SCJ scope memory model we simplify the actual scope handling code so it is included in a single, implementation private class `Memory` [39]. The `Memory` class implements the three memory areas as defined by SCJ: immortal, mission, and private memories. The RTSJ classes for these memory areas delegate their functionality to the `Memory` class. As in SCJ there is a clear hierarchy in the scopes, compared to the more complex cactus stack of scopes in RTSJ, the implementation of `Memory` can manage this nesting by reserving backing store from a `Memory` objects that *represents* the outer scope and the remaining backing store.

Preemptive scheduling of periodic handlers can be achieved basically by implementing a scheduler and using a programmable timer interrupt. As JOP and HVM support first level interrupt handlers written in Java [24], the scheduler is simply such a first level interrupt handler registered to handle the timer interrupt.

5.1. *Safety-Critical Java on JOP*

JOP is a Java processor directly executing Java bytecodes. Therefore, Java is the native language for the processor. There is no additional language layer or operating system involved in the JOP runtime. The JVM is the operating system, being responsible for thread scheduling, locking, and memory management. Therefore, we implemented the SCJ infrastructure directly in Java [40]. Currently the implementation on JOP supports level 0 and level 1 of SCJ.

The original design of JOP targeted embedded and real-time applications. Therefore, JOP did not support the standard Java thread model, but a model with periodic threads and event handlers [15]. This original threading model of JOP is very similar to the periodic and aperiodic event handlers in

DOI: 10.1002/cpe

SCJ. Therefore, we used that already available and well-tested infrastructure.[§] SCJ handlers simply delegate to the JOP real-time threads and software event handlers.

The scheduler of JOP, written in Java, implements a standard fixed-priority preemptive scheduler. A programmable timer is reprogrammed on each scheduling decision to fire when a higher priority thread will be ready in the future. Handling of locks in the current version of JOP is implemented by disabling of interrupts, including the timer interrupt. This is a simplification relative to the SCJ standard, but is very efficient if only short synchronized sections are used. In order to relax this restriction on synchronized sections, especially for multicore versions of JOP, hardware support for locking has been developed (see next Section).

The number of cores in JOP is configurable [31]. SCJ level 1 defines partitioned scheduling, which means each thread is pinned to exactly one core. Partitioned scheduling is easier to analyze with regards to properties of real-time systems and simpler to implement. Each core has its own timer interrupt and executes its own scheduler. Multicore support is only defined for SCJ levels 1 and 2. However, in an experimental setup we have also explored the static schedule of level 0 with a multicore version of JOP [32]. The challenge is to include the resource requests into the static schedule generation.

JOP and HVM use a private class `Memory` for all RTSJ inherited and in SCJ newly defined scope classes [39]. This results in mostly empty classes from RTSJ. If the SCJ standard could cut the inheritance of `ManagedMemory` from the RTSJ classes, it would be possible to avoid those dead code classes.

As our implementation has been done with an earlier version of the SCJ specification, the implementation does not yet support the newly added features, such as one-shot event handlers. We have explored an experimental version of user-defined clocks [41], but have not yet updated that implementation to the current SCJ specification.

5.2. *Safety-Critical Java on HVM*

The SCJ implementation on top of HVM [42, 23] has the architecture shown in Figure 2, with a minimal hardware interface layer specified in the `vm` interface.

This interface is divided into three parts:

- `Memory` that controls the memory allocation
- `Process`, `Scheduler`, `InterruptHandler`, and `Monitor` that define the interfaces to process, process scheduling, context switch, and synchronization, and
- `RealtimeClock` that defines clock specific methods.

The SCJ implementation on HVM is a bare metal implementation, which also takes advantage of hardware objects and other hardware near features. Thus, it has no native function layer.

The implementation strategy for the SCJ classes is to use classes from the `vm` interface. Here delegation is used:

- class `MemoryArea` has a field `delegate` of type `vm.Memory`
- class `ManagedEventHandler` uses an infrastructure class `ScjProcess` that has a field `delegate` of type `vm.Process`, and
- class `PriorityScheduler` delegates to an infrastructure class `PrioritySchedulerImpl` that implements `vm.Scheduler`.

SCJ supports method synchronization for shared resources. This is implemented in an SCJ infrastructure class `Monitor` which extends the abstract `vm.Monitor` class. As for the level 2 implementation this `Monitor` class also implements the `wait`, `notify`, and `notifyAll` methods. Furthermore, this infrastructure class is used for implementing the priority ceiling emulation that SCJ requires to avoid priority inversion.

[§]An application using that infrastructure is in industrial use by the Austrian railways since 2004 [33]. Several trains receive help to coordinate usage of single track lines by a JOP based system on a daily basis.

DOI: 10.1002/cpe

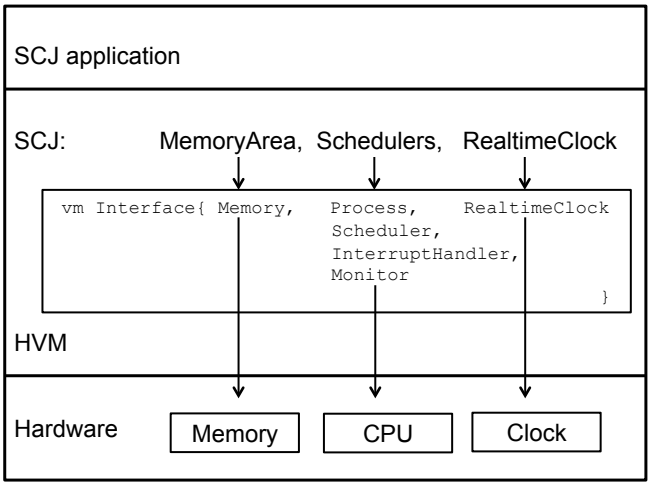

Figure 2. SCJ architecture with the vm Interface to HVM

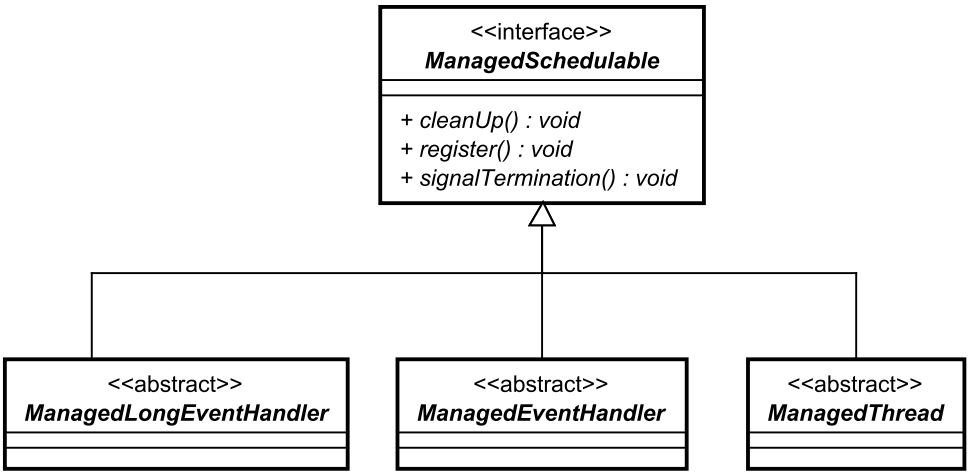

Figure 3. `ManagedSchedulable` classes

Level 0 and level 1 were implemented initially [42], because these levels target applications running on resource constrained embedded platforms. As a step towards supporting more resourceful (possibly multicore) platforms, level 2 has been added [43]. This introduces the following concepts:

- managed threads, scheduled by the priority scheduler,
- `wait`, `notify`, and `notifyAll` for managed schedulable objects,
- nested mission sequencers, and
- a modified memory management model.

To implement managed threads, the infrastructure is changed to support both threads and event handlers, i.e., objects of type `ManagedSchedulable`, see Figure 3, since the initial level 0 and 1 infrastructure only supported event handlers. The threads are modeled in the `ManagedThread` class. The scheduling of threads is implemented by refactoring the priority scheduler. The scheduling of threads is similar to that of aperiodic event handlers: (1) if the thread is preempted during executing, it will be put back into the ready queue and wait to be scheduled again; (2) if the thread finishes its execution, the `cleanUp` method of the thread will be called and its private memory area will be removed.

DOI: 10.1002/cpe

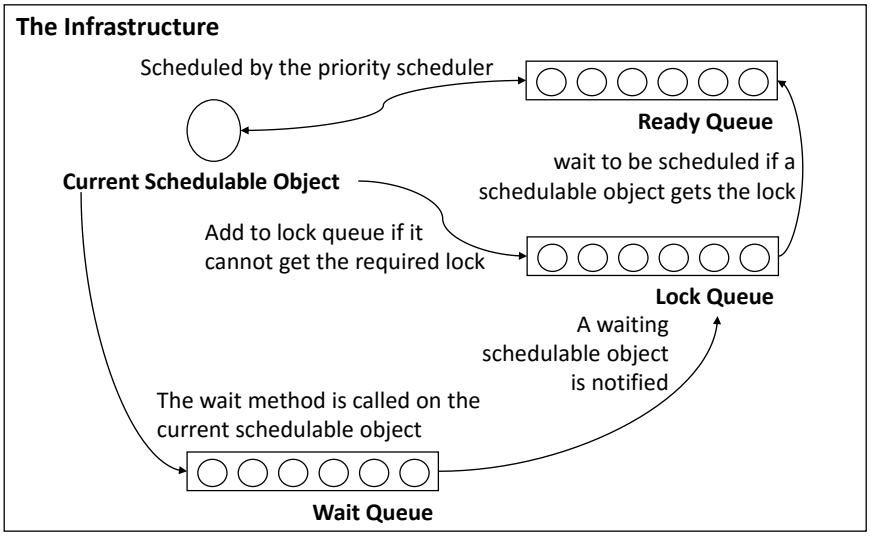

Figure 4. Wait and notify facility (for level 2)

To support `wait` and `notify`, the initial locking mechanism is modified and a priority queue is introduced: the lock queue, shown in Figure 4. If the current thread requests to access a shared resource and fails, it will be placed in the lock queue. When the resource becomes available, the highest priority thread that requests the resource in the lock queue will get the lock and be placed in the ready queue to be scheduled.

To implement the `wait` and `notify` facility, another priority ordered queue, i.e., the wait queue is introduced and the class `PriorityScheduler` is extended with two new methods `void wait(Object target)` and `void notify(Object target)`. When the current thread calls `wait` or `notify` inside a synchronized method of a shared object, these calls are delegated to the scheduler, passing the shared object (`target`) as actual parameter. Then the scheduler can rearrange its internal queues. In the case of `wait`, the scheduler will yield and switch to another thread. In addition, the current thread will release the lock and be placed into the wait queue. In the case of `notify`, the highest priority thread that waits for the lock in the wait queue (if any) will be placed into the lock queue and begin to compete for the lock. The calling thread continues until the next reschedule point.

The feature of nested mission sequencers enables missions to create inner mission sequencers so that missions can be executed concurrently. The creation of a nested mission sequencer is the same as that of any schedulable objects: the sequencer should be registered during the mission initialization phase. Once the mission enters into the execution phase, the sequencer will be scheduled to execute and its child missions will be created. After the child missions are terminated, the sequencer will be cleaned up by the infrastructure and its associated memory areas will be removed.

To track all the activated missions in the infrastructure, a static mission set is modeled in the `Mission` class to store all the activated missions. In addition, each mission maintains a set that contains all the schedulable objects it created so that each individual schedulable object can be tracked. By doing so, the time complexity for getting a specific schedulable object remains $O(1)$. This facility is illustrated in Figure 5.

Finally, the initial implementation of the level 0 and 1 memory management model only supported missions that are executed sequentially, since it was not complete with respect to the model defined in the SCJ specification. For level 2 applications, adding full support for the *backing store* concept completed the memory management implementation. In the final implementation, a memory area can also act as a backing store: it can contain other memory areas. New memory areas will be allocated in the backing store space of a specific memory area. This feature avoids fragmentation in the case of nested mission sequencers. In addition, the feature of nested private memory areas

DOI: 10.1002/cpe

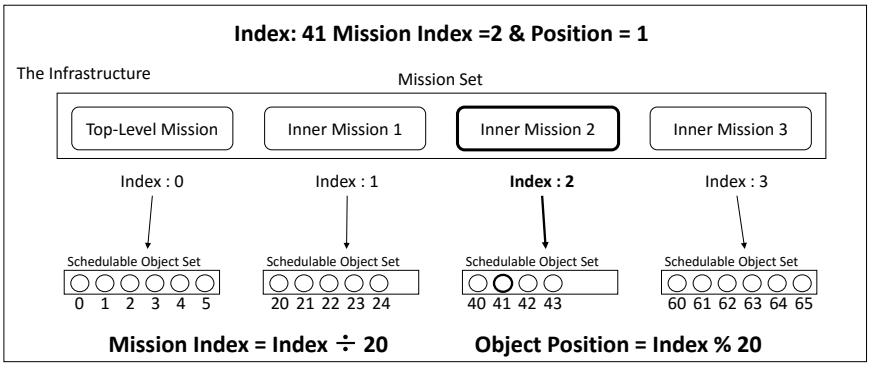

Figure 5. Nested mission sequencers (here, at most 20 schedulable objects per mission)

and the memory resizing mechanism are implemented. With the support from backing stores, each mission sequencer will be allocated a whole block of memory, which contains its private memory, mission memory and a backing store space (where the handlers memory will be allocated).

6. HARDWARE SUPPORT

Designing a Java processor provides the freedom to add hardware features that support operations of the SCJ. Hardware support can make those operations more efficient or, what we consider more important, more time-predictable. We picked scope checks and support of Java locks, as examples to explore hardware support for SCJ.

6.1. Scope Checks

Objects allocated in different scopes need to follow strict assignment rules. Objects in longer-lived scopes are not allowed to have a reference to an object allocated in a shorter-lived scope. Or in other words, an object is not allowed to reference objects allocated in an outer scope. Otherwise this assignment could lead to a pointer to an undefined data structure. Therefore, an SCJ compliant implementation needs to check all reference assignments, which might result in a considerable runtime overhead. Reference assignments need to be checked when executing bytecodes `putfield` for a reference field, `putstatic` for a reference field, and `aastore`.

SCJ, in contrast to RTSJ, supports only a simple nesting relation that results in a unique level for each scope. This scope level can be stored in the object (header) at creation time and will never change. A simple comparison of the levels implements the scope check. As a further optimization the scope level can also be encoded in some bits of each object reference.

The above described assignment check with scope levels can be implemented in any software JVM and as well in hardware on JOP. We implemented this check in hardware in JOP in the memory management unit (MMU). That unit performs all memory related operations, such as access to object fields. With the scope encoded in the reference itself we added the level comparison to the MMU as an additional state in the state machine that implements the memory-related bytecodes. If the assignment is illegal, the MMU triggers an interrupt and the interrupt handler then throws an `IllegalAssignmentError`.

The additional state in the MMU for those bytecodes increases the execution time of the related bytecodes by a single clock cycle compared to the implementation without assignment checks. Compared with a software implementation, the hardware solution is about 14 times faster. The additional hardware (extending the state machine and adding a comparator) are very minimal, just 4 % of the MMU.

DOI: 10.1002/cpe

6.2. Multicore Locking Unit

Java allows every object to serve as a synchronization lock. Locks are accessed when using either synchronized blocks or synchronized methods. SCJ only allows objects with synchronized methods function as locks.

JVMs usually implement locks using low-level atomic operations, such as compare-and-swap. In addition, some form of lock tracking is used such as hash maps or, more commonly, in the header of the objects themselves. There are however issues with these approaches. Compare-and-swap requires shared memory, which is limited by memory arbitration, and, in the case of time-division multiplexing, potentially increases the length of arbitration slots. Hash maps do have amortized constant lookup times, however they are not viable in real-time systems. The object header provides constant time lookup, but does require space in the object header, potentially increasing the size of all objects.

JOP uses a single, shared global hardware lock. This avoids additional memory arbitration, but reduces all locks, even noncontending ones, to this single lock. To avoid sharing a single lock for all critical sections we worked on a multicore hardware locking-unit [44]. The locking-unit is based on a content-addressable memory (CAM) hardware unit, but extends it in important ways. A CAM stores addresses of objects currently used as locks. A core supplies the address of an object that it wishes to lock on and the CAM checks this address with all existing addresses. In the case where the address already exists, a software routine enqueues the core and blocks it until all cores ahead of it in the queue have released the lock. Otherwise, a new lock entry is created for the supplied address and the core is allowed to continue execution. Access to the CAM is serialized using the global lock.

With the CAM, the address-to-queue mapping can be done without the need for a lock entry in objects or the need for a software hash-map. However, if the queuing is done in software, memory arbitration will still be an issue. To avoid this, the locking-unit implementation merges the CAM, the queues, and the global lock into a single multicore locking unit. Placing the queues in hardware enables the unit to be accessed through core-local microcode, instead of Java code in shared memory. This removes any memory arbitration in the locking procedure. Serializing lock requests is still necessary. However, merging the global lock with the unit means that only a single unit has to be accessed, reducing the number of software steps.

Benchmarks show that this new unit performs at least as well as the global lock and, in the case of heavy lock contention or large noncontending critical sections, far better [45].

7. ANALYSIS TOOLS

During the lifetime of the CJ4ES project, two tools have been created for aiding programmers developing for the SCJ specification.. The tools are intended to help developers with some of the challenges arising from the scoped memory model of SCJ, namely checking that no runtime exceptions due to violations of memory safety can occur and calculating a sound approximation of the maximum amount of memory needed for a given SCJ application. These tools provide static guarantees that no memory related exceptions would be thrown at runtime, an essential property for certification of safety-critical applications.

The tools are both developed using the T.J. Watson Libraries for Analysis (WALA) [46]. In addition to being well integrated in the Eclipse development environment, WALA includes highly scalable, extensible, and configurable, state-of-the-art pointer analyses for Java bytecode. In particular, a context sensitive points-to analysis is included that supports user definable contexts.

7.1. Memory Safety

While the explicit and manual memory management defined by the scoped memory model of SCJ (see Section 3) affords programmers a high degree of control, it also introduces the possibility of *dangling references*, which are avoided within SCJ assignment rules. SCJ only allows references to objects that live in a scope with the same or longer lifetime as that of the reference, i.e.,

DOI: 10.1002/cpe

Table I. Results of Memory safety analysis

Test case	LOC	Library	Application	IA	RT	FP
scjminepump	1476	275050	18672	0	0	0
scjminepumplog	1500	275050	20664	1	1	0
pmFFTcpResult	523	282915	11660	0	0	0
InOutParameter	162	301583	6440	1	2	1
RepRap	2636	277426	42258	2	2	0
ScjL1Pacemaker	845	293941	33278	0	0	0
MiniCDj	3366	287463	231243	-	-	-
Quicksort	388	278663	7846	0	0	0
Fast-MD5	985	278663	15649	0	0	0
Thruster-Engine	426	276617	8483	0	0	0

references on the current scope stack can only point *away* from the top of the stack and towards older scopes. Violations of this rule result in a `IllegalAssignmentError` exception being thrown at runtime. For safety-critical systems, especially for systems that must be certified, runtime exceptions are usually unacceptable and developers of such systems must therefore show that no runtime exceptions will be thrown by the system. There are several approaches for giving such guarantees at compile-time, including manually adding *annotations* and performing a static *memory safety analysis*. The former approach requires developers to (manually) add annotations to key elements of the program, indicating the intended scope of that element. Using these annotations, it is possible for an automated tool to verify that the intended scope use cannot lead to a memory safety violation [47, 48].

Related work, such as work on ownership types [49], demonstrate that it is possible to eliminate runtime memory checks. Work on amortized resource analysis using separation logic offers a more formal method of resource analysis by embedding logic annotations, e.g., in the form of pre- and post-conditions. Recent work has demonstrated how such methods can be applied to traditional Java programs [50, 51]. Currently, this work still has some limitations, e.g., it is only possible to handle bounds on resource usage that are linear in the size of data structures.

In the latter approach, a static analysis is performed that tracks all the possible scope stacks across the entire program and uses this to verify that no reference can ever point to an object in a shorter lived memory area and thus, that no memory safety violations can occur in the program.

CJ4ES project members have developed a memory safety analysis and implemented a prototype tool that is able to identify potentially illegal memory assignments, i.e., assignments that may lead to a memory safety violation [52]. The analysis was designed as an extension of a context-sensitive points-to analysis where an abstract representation of the current scope stack is used as the context for the analysis. This design can be directly implemented using the points-to analysis framework included in WALA [46], which is parameterized over user-defined contexts. The analysis is sound and thus will find all possible memory safety violations, but it may report false positives, i.e., warnings of potential memory safety violations that can never occur in a real execution of the program. However, on the suite of applications the analysis was tested on, only few false positives were reported. The tool was used to report on five test cases. Since then we have updated the tool to version 0.93 of the SCJ specification and also doubled the number of test cases.

The results of the experiments can be seen in Table I. The table shows the result of running the analysis on 10 test cases. 'LOC' is short for Lines of Code. The columns 'Library' and 'Application' list the sizes of the SCJ library and the application in number of bytecodes. 'IA' is short for Illegal Assignments, 'RT' is short for reported by tool and 'FP' is short for false positives. The source code of the test cases, except the RepRap and MiniCDj test cases, can be found at `https://github.com/jop-devel/apps`. The source of the RepRap and MiniCDj test cases can be found together with the JOP SCJ JVM. The latest version of the RepRap and MiniCDj test cases was used for these experiments. The ScjL1Pacemaker test case was based on an SCJ level 2

DOI: 10.1002/cpe

pacemaker application, which was refactored as an SCJ level 1 application [53]. The Quicksort, Fast-MD5 and Thruster-Engine were taken from the ScjChecker project but were adapted to the current version of SCJ.

Version 0.93 of the SCJ specification contains two relevant changes for the memory safety analysis. First, the new specification clarifies the initialization phase and secondly, through API changes, it is no longer possible for application programmers to get a reference to a memory area. These changes simplify analyses for identifying illegal memory assignments, as the analyses no longer need to track variables storing references to memory areas. To the extent needed we have updated test cases to adhere to the API changes.

In order for the analysis to be sound, it needs to analyze both the SCJ application and the underlying SCJ implementation. However, in some cases developers may want to analyze applications separately from any particular SCJ implementation, e.g., while developing an SCJ application that will be used in combination with multiple SCJ implementations. In order to handle such use cases in our analysis, we have implemented prototype support for only analyzing the SCJ application by the use of an incomplete stub library. This analysis is unsound, by nature, as potential allocation in the runtime environment itself is not modeled. Indeed, in one of our test cases, the InOutParameter test case, we observed that one issue resulting from clever usage of the `StringBuilder` class, which should have been a false positive, was not reported when only analyzing the application. Missing potential errors in a safety critical system is undesirable at best and for this reason, safety-critical applications should be analyzed in conjunction with an appropriate SCJ implementation. The implementation is based on stub classes and methods as well as support in the analysis for skipping analysis of unimplemented methods. The prototype implementation was tested on a subset of the test cases [52]. Adding further stub classes and improving support for unimplemented methods in the analysis can support more test cases.

In addition to statically verifying that a program cannot violate memory safety, the results of our memory safety analysis can also be used for generating scope annotations that can be added to the program. This is useful, e.g., for documenting the results of the analysis in a fashion that can be automatically checked by other tools [47, 48], or even be used for understanding a program developed by a third party.

Our analysis is closely related to the data-flow analysis developed by Siebert [54] for RTSJ, but exploits the simpler structure of the SCJ memory model (see Section 3.2) to achieve both high precision and good performance. Similar results where found by Marriott and Cavalcanti [55]. The authors developed a static analysis that verifies that an application does not contain illegal memory assignments by using an abstract language and inference rules. The transformations used in that work are backed by a formal proof of correctness. In a slightly different direction, a hardware implementation of the scope checking was presented by Rios et al. [56], showing that such checking can be done very efficiently for SCJ in hardware due to the simple stack structure of the memory scopes in use.

7.2. Worst-Case Memory Consumption

As noted in Section 3.2, the scoped memory model requires programmers to provide explicit bounds on the concrete amount of memory needed for the backing store of event handlers and the various memory areas in general. Calculating these manually is difficult and error prone at best, since the developer must track both memory usage and the current scope stack through all possible execution paths in the program and map this information back onto a specific platform with particular requirements for memory usage and alignment.

In order to assist the programmer and automate the task of calculating these memory bounds, we have developed a *worst-case memory consumption (WCMC) analysis* [57]. The WCMC analysis is based on the *implicit path enumeration technique* (IPET) to reformulate the WCMC problem as an integer linear programming problem (ILP). This is similar to an approach adapting an existing WCET analysis tool to search for maximum memory allocations instead of searching for maximum number of execution cycles [58].

DOI: 10.1002/cpe

The problem of cost analyses for Java bytecode has previously been studied in COSTA [59], a tool for cost and termination analysis. Compared to the WCMC analysis [57], COSTA derives closed-form expressions dependent on input variables and uses a recurrence relation system in contrast to the WCMC [57] analysis that is based on ILP. The recurrence relation system also enables inference of upper bounds on recursion. The WCMC analysis has been implemented on the framework of the memory safety analysis described in the previous section, using WALA [46] and a standard ILP solver, such as lp_solve. ¶

8. APPLICATIONS

To explore the expressiveness of SCJ and also to have test cases, we have implemented several applications and small test cases on top of SCJ.

8.1. RepRap

A RepRap 3D printer [60] melts plastic and extrudes it in 3 dimensional spaces according to printing instructions (G-codes). This enables it to construct or "print" 3D objects in plastic. Figure 6 shows the RepRap hardware, an FPGA board containing JOP, and the custom-made interface board between the FPGA board and the electronics of the RepRap printer.

The printer uses five stepper motors: two for the X and Y dimensions, two for the Z dimension, and one for extruding plastic. The printer also uses a resistor for heating. We consider the printer a safety-critical application as it employs physical movement and heating above 200°C.

The system consists of the physical printer, an electronic interface board, an SCJ RepRap controller running on JOP on an FPGA, and a PC. The PC generates G-codes from a 3D image. The G-codes are sent over a serial line to the controller. The controller enqueues and executes the instructions and controls the motors and the heating element in the process through the interface board.

The controller consists of the periodic event handlers `HostController`, `CommandParser`, `CommandController`, and `RepRapController`. The `HostController` handles all communication with the host (PC). When the `HostController` sees an instruction delimiter, the previously received characters are sent to the `CommandParser`, which parses the instruction and enqueues the corresponding Command object in the `CommandController`. The `CommandController` executes the commands in FIFO order, with some commands modifying control parameters in the `RepRapController`, such as position and temperature, and others writing information, such as current temperature and command status, to the host through the `HostController`. The `RepRapController` controls the motors and temperature according to control parameters. The motors are stepped towards a position in a timed manner, ensuring that all motors reach their goal at the same time regardless of the distance.

In order to ensure that the event handlers are schedulable we analyze the handlers' WCET and maximum blocking time using JOP's WCET tool [8]. The results are shown in Table II. The event handlers are listed from highest to lowest priority. The `RepRapController` needs a short period to ensure that printing is not too slow, whilst the `HostController` needs a short period to ensure that all characters are properly received from the host according to the specified bit rate.

The WCET column shows the worst-case execution time of each event handler's `handleAsyncEvent` method. The last column shows the maximum time a handler is blocked by another handler because of synchronization/locking.

The response time analysis [61] (see also [62, p. 490]) states that a set of periodic tasks is schedulable with fixed priority scheduling if, for each task, the response time of a given task is

¶http://lpsolve.sourceforge.net/5.5/

DOI: 10.1002/cpe

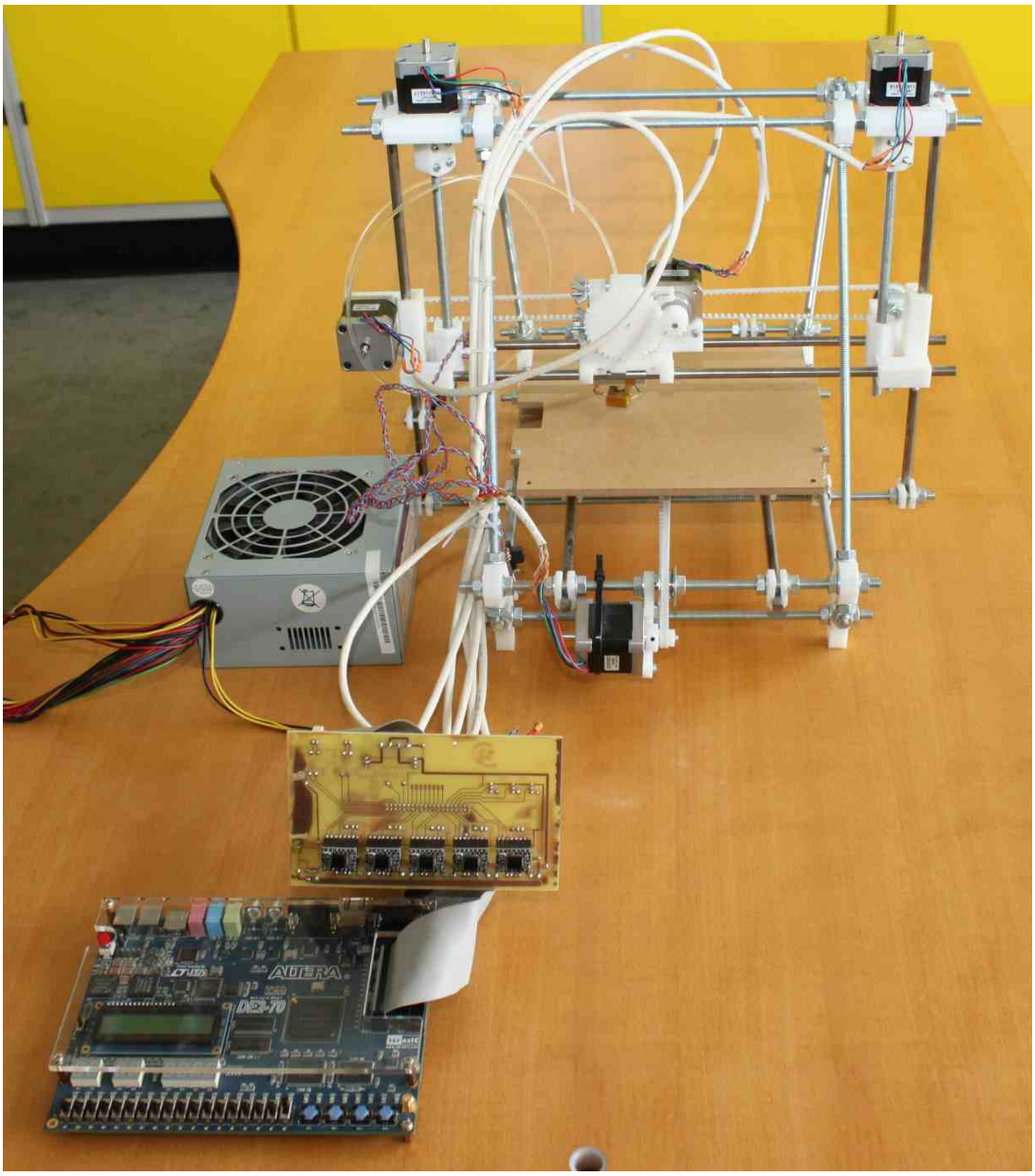

Figure 6. RepRap setup without the host

Table II. WCET for the PeriodicEventHandlers

PEH	Priority	Period (ms)	WCET (ms)	Max. blocked (ms)
RepRapController	4	1	0.072	0.002
HostController	3	1	0.426	0.153
CommandController	2	20	0.914	0.153
CommandParser	1	20	3.578	0.153

lower than its period (and thereby deadline). More specifically, a task is schedulable if it satisfies

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j$$

DOI: 10.1002/cpe

where i is the current task, $hp(i)$ are tasks with a higher priority than i, C_i is the task's worst-case execution time, T_j is a higher priority task's period, and B_i is the task's maximum blocked time. This can be solved using the recurrence relationship

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{w_i^n}{T_j} \rceil C_j$$

We performed the response time analysis for each of our event handlers and presenting the numbers in Table II. All the response times are lower than the corresponding periods, showing that the handlers are schedulable.

8.2. *Cubesat Space Protocol*

One of the partners of the CJ4ES project, GomSpace ApS, is a company that specialises in developing and building small satellite platforms, so-called cubesats and nano-satellites. The individual components of a cubesat communicate via a communications protocol originally developed at Aalborg University specifically for use on board satellites: the Cubesat Space Protocol (CSP). We have implemented that protocol and on top of this protocol a watchdog that checks the health of the connected components. The watchdog performs a reset when a ping request is not answered [57]. This small application and the protocol implementation use two periodic event handlers and an interrupt handler for incoming packets. We tested this application and the implementation of the Cubesat space protocol by connecting a JOP system, implemented in an FPGA, to a onboard computer of a Cubesat developed by GomSpace.

8.3. *External Applications*

Within the project we also searched for and explored SCJ example applications from other research groups. When needed, we adapted them to the current version of the specification. The updated code is available at the project's repository.

The miniCDj benchmark is a reduced version of the CDx benchmark [63]. This benchmark generates simulated radar frames containing airplane positions and calculates possible collisions between those simulated radar frames. We have used two versions of it: the original implementation which uses a cyclic-executive (level 0), and a parallel version (level 1) [64] adapted at the University of York, which divides the task of looking for collisions into a fixed number of AEHs. In both cases, we have updated the benchmarks to the current version of the SCJ specification as both of them were based on version 0.76.

The technology compatibility kit (TCK) described in [65] is an early work done to develop a TCK for SCJ. It is however based on version 0.76 of the specification. We have updated that TCK to the current version and used it to test the level 0 and level 1 features of the SCJ implementation of JOP.

9. LESSONS LEARNED

Development in the CJ4ES project provided a good insight into the SCJ proposal and how to use Java for safety-critical systems. One of the strengths of SCJ is that its simpler subset of the RTSJ can simplify the task of developing analysis tools. Even though the SCJ specification incorporates many annotations intended to implement static analysis tools [48], recent research has shown that it is indeed feasible to develop analysis tools that do not rely on such annotations [52, 57]. Moreover, a simplified subset of a language is easier to express in a formal way and therefore it is expected that implementing formal analysis tools for SCJ is a feasible task. There is indeed research in this direction that uses the Java modeling language (JML) [66, 67, 68] to formally specify and verify properties of SCJ. From these arguments it follows that two of the generic challenges for safety-critical systems, namely those related to formal verification and development, time and effort can be addressed with the new technology proposal of SCJ.

DOI: 10.1002/cpe

9.1. Application Development

From an application developer perspective, SCJ at levels 0 and 1 provides a familiar programming model for real-time systems developers. However, one of SCJ's weak points is the absence of a garbage collector, which has made standard Java so popular. In our experience, the use of the scoped memory model initially leads to constant cycles of testing and debugging, as we needed to keep track of where objects are allocated in order to avoid illegal reference assignments. Correct use of the scoped memory model is indeed the most difficult part when developing SCJ applications. This change of mindset is most likely the biggest barrier that Java developers may face. However after getting used to the scoped memory model, development of SCJ applications becomes easier and more intuitive.

A factor that limits the development of reusable software in SCJ comes from the restrictions that the scoped memory imposes to library code. Libraries in Java offer a working and tested solution to ease the development of reusable software. However, in SCJ, safe use of library code is restricted to the boundaries of scoped memories as any temporary or auxiliary object created by the library code can be safely allocated and referenced only from within the boundaries of scoped memories. The use of library code becomes more difficult when crossing scope boundaries, i.e., calling methods of library classes from different scopes, either as a result of the developer's intent or because the library classes make calls to other library classes (which is very common). If it is the intent of the developer to use library code in mixed scoped memory contexts, then the developer has to be aware of the memory allocation behavior of the code. Such behavior is not documented in standard library code and therefore, library code developed for SCJ must document, in addition to its functional behavior, its memory allocation behavior in terms of temporary objects and the expected allocation context of arguments and returned objects.

Despite the limitations mentioned, SCJ is a good choice for the development of embedded safety-critical systems as it brings the possibility to implement hard real-time systems using a safer language than C (strongly typed without pointer arithmetic). Once the developer is familiar with the use of the scoped memory model, development of SCJ applications becomes easier.

9.2. Memory Area Sizing

In general our experience suggests that the scoped memory model fits a relevant class of resource constrained embedded devices well where no heap allocated data is used and therefore no garbage collector needed. Instead, memory resources get allocated statically. But it can be difficult to safely assess how much private memory to allocate for the mission sequencer, mission memory, immortal memory, and handler stacks. The default values for those areas will allow many programs to run without running out of memory but they are over-conservative and allocate far more memory than is actually required. The memory configuration part of a typical SCJ program may look like this,

```
Const.OUTERMOST_SEQ_BACKING_STORE = 140 * 1000;
Const.IMMORTAL_MEM = 50 * 1000;
Const.MISSION_MEM = 35 * 1000;
Const.PRIVATE_MEM = 2 * 1000;

storageParameters_Sequencer =
  new StorageParameters(
    Const.OUTERMOST_SEQ_BACKING_STORE,
    new long[] { Const.HANDLER_STACK_SIZE },
    Const.PRIVATE_MEM,
    Const.IMMORTAL_MEM,
    Const.MISSION_MEM);

storageParameters_Handlers =
  new StorageParameters(
    Const.PRIVATE_MEM,
    new long[] { Const.HANDLER_STACK_SIZE },
    Const.PRIVATE_MEM,
    0,
```


DOI: 10.1002/cpe

```
0);
```

The first four statements overwrite the default values to decrease memory requirements. The next two statements create, as a convenience, two `StorageParameters` objects to be used later when creating the sequencer and the handlers. This configuration can be difficult to get right for most developers new to SCJ. At least two ways exist to solve this problem (1) static program analysis can suggest a conservative, but tight, estimate for the size of each memory area [58, 57] and (2) profiling features can be added to the VM to report—after program execution—how much memory was actually used in the individual memory areas. The latter requires that the program can run on the target with the overly conservative default values or that the program can be run in a simulated environment on a platform where memory resources are ample. The latter method is supported by the HVM. For smaller SCJ programs for embedded systems, the majority of the memory requirements stem from the stacks, of which there is usually one unique for each handler. Determining how large the stacks manually is difficult. There is need for a tool to determine the maximum stack size. If only the Java stack size needs to be known, as for the JOP version of SCJ, the maximum stack size can be statically determined.

9.3. RepRap Lessons Learned

From a Java developer's point of view, SCJ provides a familiar setting with a lot of the behavior being similar to standard Java. However, the main difference, and also the main issue, is the use of scoped memory. In Java, a developer mostly has to consider object creation with regards to object-oriented principles. With scoped memory another layer is added to the consideration. The issue with using scopes and implementing scope safe libraries has been experienced with the development of the RepRap use case and was summarized in Subsection 9.1.

Another issue is the performance of low-level operations. The object-oriented nature of SCJ allows programs to be well structure and easily maintained. However, this also means added performance overhead. Controlling motors can require a handler to execute several thousands of times per second. In this case creating new result objects for calculations can infer too much overhead. This can somewhat be overcome by only using primitives and avoiding library code. However, the object-oriented overhead can still make low-period handlers infeasible. In our experience SCJ is therefore good for higher abstraction levels and overall control, whereas high-performance, low-level operations are designated to other systems.

9.4. Implementation

The dependencies on RTSJ also make the implementation of the profile more challenging. The consequences are that: (1) information of implementation classes needs to be shared over package boundaries (RTSJ and SCJ live in their own packages), (2) most RTSJ classes become greatly restricted and possibly empty, and (3) complex implementation features of RTSJ will also be part of an SCJ implementation that runs on top of the RTSJ. For safety-critical applications that need to be certified, this dependency will increase the effort and cost and there will be much code that is inactive. Such deactivated code is considered a risk in safety-critical applications and will have to be removed by static analysis tools. This is a feasible approach, and may have to be applied anyway to remove other inactive code in a specific application. Yet, a simpler profile that follows a bottom-up approach would be desirable in order to avoid inheriting unnecessary complexity (see for example [12, 5, 6]).

9.5. Teaching Material

The SCJ programming model is different from normal Java and apart from configuring memory it can also be difficult for the novice SCJ programmer to properly plan and start the execution of the sequencer and handlers. Simple tutorials and guides on how to create simple SCJ programs are missing from the public domain. Also teaching material, like a book for students, could really benefit the acceptance and success of the technology.

DOI: 10.1002/cpe

10. CONCLUSIONS

The upcoming standard for safety-critical Java (SCJ) will help to develop safety-critical applications in Java. To explore the strengths and weaknesses of SCJ we implemented SCJ on two different platforms: in Java on a Java processor and on a hardware near virtual machine. To assist the SCJ programmer we developed a memory safety analysis tool. Finally we explored SCJ with library developments and example applications. Our conclusion on SCJ is that is a challenging framework to write a program against it, especially getting the usage of scope memories correct. However, we think this restrictive programming model fits well for safety-critical applications that need to be certified. This paper summarizes the developments and key contributions from the three-year project "Certifiable Java for Embedded Systems (CJ4ES)".

Source Access

The two presented Java processor JOP and the software JVM HVM, the implementations of safety-critical Java on top of them, and the two analysis tools are provided in open source. The Java processor JOP and the SCJ implementation are hosted at GitHub at https://github.com/jop-devel/jop. The hardware near virtual machine is available from https://github.com/scj-devel/hvm-scj, which includes a version of SCJ as well. The source of the two analysis tools is also available from GitHub at https://github.com/scj-devel/tools.

ACKNOWLEDGEMENT

This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and has received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

REFERENCES

1. Bollella G, Gosling J, Brosgol B, Dibble P, Furr S, Turnbull M. *The Real-Time Specification for Java*. Java Series, Addison-Wesley, 2000.
2. Henties T, Hunt JJ, Locke D, Nilsen K, Schoeberl M, Vitek J. Java for safety-critical applications. *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, 2009.
3. Schoeberl M. Jop: A java optimized processor for embedded real-time systems. PhD Thesis, Vienna University of Technology 2005.
4. Schoeberl M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture* 2008; **54/1–2**:265–286, doi:http://dx.doi.org/10.1016/j.sysarc.2007.06.001.
5. Schoeberl M, Sondergaard H, Thomsen B, Ravn AP. A profile for safety critical Java. *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, IEEE Computer Society: Santorini Island, Greece, 2007; 94–101, doi:10.1109/ISORC.2007.9.
6. Bøgholm T, Hansen RR, Ravn AP, Thomsen B, Søndergaard H. A predictable java profile: rationale and implementations. *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM: New York, NY, USA, 2009; 150–159, doi:10.1145/1620405.1620427.
7. Bogholm T, Kragh-Hansen H, Olsen P, Thomsen B, Larsen KG. Model-based schedulability analysis of safety critical hard real-time Java programs. *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, ACM: New York, NY, USA, 2008; 106–114, doi:http://doi.acm.org/10.1145/1434790.1434807.
8. Schoeberl M, Puffitsch W, Pedersen RU, Huber B. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience* 2010; **40/6**:507–542, doi:10.1002/spe.968.
9. Schoeberl M, Dalsgaard AE, Hansen RR, Korsholm SE, Ravn AP, Rivas JRR, Strøm TB, Søndergaard H. Certifiable Java for embedded systems. *Proceedings of the 12th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2014)*, ACM: Niagara Falls, NY, USA, 2014; 10–19, doi:10.1145/2661020.2661025.
10. Nilsen K, Lee S. Perc real-time api (draft 1.3). newmonics July 1998.
11. Nilsen K, Carnahan L, Ruark M. Requirements for real-time extensions for the Java platform. Available at http://www.nist.gov/rt-java/ September 1999.
12. Puschner P, Wellings A. A profile for high integrity real-time Java programs. *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, Magdeburg, Germany, 2001; 15–22.
13. Kwon J, Wellings A, King S. Ravenscar-Java: A high integrity profile for real-time Java. *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, ACM Press: Seattle, Washington, USA, 2002; 131–140, doi:http://doi.acm.org/10.1145/583810.583825.

14. Burns A, Dobbing B, Romanski G. The Ravenscar tasking profile for high integrity real-time programs. *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, Springer-Verlag, 1998; 263–275.
15. Schoeberl M. Restrictions of Java for embedded real-time systems. *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, IEEE: Vienna, Austria, 2004; 93–100, doi:10.1109/ISORC.2004.1300334.
16. Søndergaard H, Thomsen B, Ravn AP. A Ravenscar-Java profile implementation. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, ACM: Paris, France, 2006; 38–47.
17. aJile. aj-100 real-time low power Java processor. preliminary data sheet 2000.
18. Plsek A, Zhao L, Sahin VH, Tang D, Kalibera T, Vitek J. Developing safety critical Java applications with oSCJ/L0. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, ACM: New York, NY, USA, 2010; 95–101, doi:http://doi.acm.org/10.1145/1850771.1850786.
19. Armbruster A, Baker J, Cunei A, Flack C, Holmes D, Pizlo F, Pla E, Prochazka M, Vitek J. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.* 2007; **7**(1):1–49, doi: http://doi.acm.org/10.1145/1324969.1324974.
20. Aonix. Perc pico 1.1 user manual. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf April 2008.
21. Nilsen K. Harmonizing alternative approaches to safety-critical development with Java. *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, York, UK, 2011; 54–63.
22. Locke D, Andersen BS, Brosgol B, Fulton M, Henties T, Hunt JJ, Nielsen JO, Nilsen K, Schoeberl M, Tokar J, *et al.*. Safety-critical Java technology specification, public draft 2011.
23. Korsholm S, Søndergaard H, Ravn A. A real-time java tool chain for resource constrained platforms. *Concurrency and Computation: Practice & Experience* September 2013; **2013**:1–25.
24. Schoeberl M, Korsholm S, Kalibera T, Ravn AP. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.* November 2011; **10**(4):42:1–42:40, doi:10.1145/2043662.2043666.
25. Cassez F, Hansen RR, Olesen MC. What is a timing anomaly? *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012)*, *OASICS*, vol. 23, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012; 1–12.
26. Schoeberl M. Design and implementation of an efficient stack machine. *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, IEEE: Denver, Colorado, USA, 2005, doi:10.1109/IPDPS.2005.161.
27. Schoeberl M. A time predictable instruction cache for a Java processor. *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, *LNCS*, vol. 3292, Springer: Agia Napa, Cyprus, 2004; 371–382, doi:10.1007/b102133.
28. Schoeberl M. A time-predictable object cache. *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, IEEE Computer Society: Newport Beach, CA, USA, 2011; 99–105.
29. Huber B, Schoeberl M. Comparison of implicit path enumeration and model checking based WCET analysis. *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, OCG: Dublin, Ireland, 2009; 23–34.
30. Huber B, Puffitsch W, Schoeberl M. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience* 2012; **24**(8):753–771, doi:10.1002/cpe.1763.
31. Pitter C, Schoeberl M. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.* 2010; **10**(1):9:1–34, doi:10.1145/1814539.1814548.
32. Ravn AP, Schoeberl M. Safety-critical Java with cyclic executives on chip-multiprocessors. *Concurrency and Computation: Practice and Experience* 2012; **24**:772–788, doi:10.1002/cpe.1754.
33. Schoeberl M. Application experiences with a real-time Java processor. *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, 2008; 9320–9325.
34. Hepp S, Schoeberl M. Worst-case execution time based optimization of real-time Java programs. *Proceedings of the 15th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2012)*, IEEE: Shenzhen, China, 2012; 64–70.
35. Nielson F, Nielson H, Hankin C. *Principles of Program Analysis*. Springer, 1999.
36. Schoeberl M, Korsholm S, Thalinger C, Ravn AP. Hardware objects for Java. *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, IEEE Computer Society: Orlando, Florida, USA, 2008; 445–452, doi:10.1109/ISORC.2008.63.
37. Luckow KS, Bøgholm T, Thomsen B. Supporting development of energy-optimised java real-time systems using tetasarts. *Work-in-Progress Proceedings of the 19th Real-Time and Embedded Technology and Application Symposium*, Philadelphia, USA, 2013; 41–44.
38. Luckow KS, Bøgholm T, Thomsen B, Larsen KG. Tetasarts: A tool for modular timing analysis of safety critical java systems. *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, ACM: New York, NY, USA, 2013; 11–20.
39. Schoeberl M. Memory management for safety-critical Java. *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, ACM: York, UK, 2011; 47–53.
40. Schoeberl M, Rios JR. Safety-critical Java on a Java processor. *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, ACM: Copenhagen, DK, 2012; 54–61, doi:10.1145/2388936.2388946.
41. Wellings A, Schoeberl M. User-defined clocks in the real-time specification for Java. *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, ACM: York, UK, 2011; 74–81.

42. Søndergaard H, Korsholm SE, Ravn AP. Safety-Critical Java for low-end embedded platforms. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, ACM: New York, NY, USA, 2012; 44–53.
43. Zhao S. Implementing level 2 of safety-critical Java. Master's Thesis, University of York 2014.
44. Strøm TB, Puffitsch W, Schoeberl M. Chip-multiprocessor hardware locks for safety-critical Java. *Proceedings of the 11th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2013)*, ACM: Karlsruhe, DE, 2013; 38–46, doi:10.1145/2512989.2512995.
45. Strøm TB, Puffitsch W, Schoeberl M. Hardware locks for a real-time java chip-multiprocessor. *Concurrency and Computation: Practice and Experience* 2015; **accepted for publication**(?):0–0.
46. T.J. Watson libraries for analysis (WALA). http://wala.sf.net/ accessed 2014.
47. Nilsen K. A type system to assure scope safety within safety-critical java modules. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, ACM: New York, NY, USA, 2006; 97–106, doi:http://doi.acm.org.globalproxy.cvt.dk/10.1145/1167999.1168017.
48. Tang D, Plsek A, Vitek J. Static checking of safety critical java annotations. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, ACM: New York, NY, USA, 2010; 148–154, doi:10.1145/1850771.1850792.
49. Boyapati C, Salcianu A, Beebee Jr W, Rinard M. Ownership types for safe region-based memory management in real-time java. *ACM SIGPLAN Notices* 2003; **38**(5):324–337.
50. Atkey R. Amortised resource analysis with separation logic. *Programming Languages and Systems*. Springer, 2010; 85–103.
51. Fenacci D, MacKenzie K. Static resource analysis for java bytecode using amortisation and separation logic. *Electronic Notes in Theoretical Computer Science* 2011; **279**(1):19–32.
52. Dalsgaard AE, Hansen RR, Schoeberl M. Private memory allocation analysis for safety-critical Java. *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, ACM: Copenhagen, DK, 2012; 9–17, doi:10.1145/2388936.2388939.
53. Singh NK, Wellings A, Cavalcanti A. The cardiac pacemaker case study and its implementation in safety-critical java and ravenscar ada. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, ACM: New York, NY, USA, 2012; 62–71.
54. Siebert F. Proving the absence of RTSJ related runtime errors through data flow analysis. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, ACM Press: New York, NY, USA, 2006; 152–161, doi:http://doi.acm.org/10.1145/1167999.1168025.
55. Marriott C, Cavalcanti A. SCJ: Memory-safety checking without annotations. *FM 2014: Formal Methods*, *Lecture Notes in Computer Science*, vol. 8442, Jones C, Pihlajasaari P, Sun J (eds.). Springer International Publishing, 2014; 465–480.
56. Rios JR, Schoeberl M. Hardware support for safety-critical Java scope checks. *Proceedings of the 15th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2012)*, IEEE: Shenzhen, China, 2012; 31–38.
57. Andersen JL, Todberg M, Dalsgaard AE, Hansen RR. Worst-case memory consumption analysis for SCJ. *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, ACM Press: Karlsruhe, DE, 2013; 2–10.
58. Puffitsch W, Huber B, Schoeberl M. Worst-case analysis of heap allocations. *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010)*, Heraklion, Crete, Greece, 2010; 464–478.
59. Albert E, Arenas P, Genaim S, Puebla G, Zanardini D. Costa: Design and implementation of a cost and termination analyzer for java bytecode. *Formal Methods for Components and Objects*, *Lecture Notes in Computer Science*, vol. 5382, de Boer F, Bonsangue M, Graf S, de Roever WP (eds.). Springer Berlin Heidelberg, 2008; 113–132.
60. Strøm TB, Schoeberl M. A desktop 3d printer in safety-critical Java. *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, ACM: Copenhagen, DK, 2012; 72–79, doi:10.1145/2388936.2388949.
61. Joseph M, Pandya PK. Finding response times in a real-time system. *Comput. J* 1986; **29**(5):390–395.
62. Burns A, Wellings AJ. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. 3rd edn., Addison-Wesley Longman Publishing Co., Inc., 2001.
63. Kalibera T, Hagelberg J, Pizlo F, Plsek A, Titzer B, Vitek J. Cdx: a family of real-time java benchmarks. *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM: New York, NY, USA, 2009; 41–50, doi:http://doi.acm.org/10.1145/1620405.1620412.
64. Zeyda F, Cavalcanti A, Wellings A, Woodcock J, Wei K. Refinement of the Parallel CDx. *Technical Report*, University of York, York, UK 2012.
65. Zhao L, Tang D, Vitek J. A technology compatibility kit for safety critical java. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, ACM: New York, NY, USA, 2009; 160–168.
66. Leavens. The Java Modeling Language (JML). http://www.eecs.ucf.edu/ leavens/JML/index.shtml Visited June 2014.
67. Schmitt P, Tonin I, Wonnemann C, Jenn E, Leriche S, Hunt JJ. A case study of specification and verification using JML in an avionics application. *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '06, ACM: New York, NY, USA, 2006; 107–116, doi:10.1145/1167999.1168018.
68. Ravn AP, Søndergaard H. A test suite for Safety-Critical Java using JML. *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, ACM: New York, NY, USA, 2013; 80–88.